Virtual Memory - Overview

- Process runs in virtual (logical) memory space – may be larger than physical memory.
- Paging can implement virtual memory.
- Which pages to have in memory?
- How much memory to allow each process?

Programmers’ View

- Program has its own virtual memory space.
  - Simple linear address space
  - Large
  - Protected
- Add to that:
  - CPU scheduling, I/O management, etc.
- You have a virtual machine for your program to run on.

Virtual → Physical

- Virtual memory must be mapped into physical memory
  - Paging or segmentation
- Efficient implementation is important
- We will consider paging

Virtual → Physical

Virtual memory must be mapped into physical memory. Options include paging or segmentation. Efficient implementation is important. We will consider paging.
### Demand Paging

- Processes not in memory are kept in secondary storage (disk).
- When required they are swapped in.
- Lazy swapping: page is only brought in when needed.
- Pager (in OS) transfers pages to/from disk.

### Page Table

- Page table records:
  - In main memory or not.
  - Main memory location.
  - Disk location.
  - Valid page or not (i.e. page has been used).

### Page Faults

- Hardware address translation looks up page table.
- Page not in memory → page fault trap.
  - Not program error, but OS does not have required page in memory.
- OS must rectify the situation.

### Handling of Page Faults

- Locate a free page frame.
  - If none, remove a page → page replacement algorithm.
- I/O to read page into frame.
- Update page table.
- Restart process at instruction causing fault.
  - Page fault trap saved state (like all traps).
- Page will now be available for instruction use.
### Page Faults

1. Reference
2. Trap
3. Page is on backing store
4. Bring in missing page
5. Update page table
6. Restart instruction

- Operating system
- Load M
- Disk
- Physical memory

### Other Considerations

- **Pure demand paging** starts process with no pages in memory.
- Programs could reference a number of pages per instruction and thereby generate a number of page faults per instruction.
  - Locality of reference reduces faults in code section.

### Support for Demand Paging

- Same hardware as paging:
  - Page table
  - Backing storage
- Plus: must be able to restart any instruction.
- For most instructions this may involve a little repeated work, but no real problems.
- For block move instructions, some addressing modes and other cases the problems are more substantial.

### Performance of Demand Paging

- Memory access time (ma) for most computers is in the range of 10 to 200 nanoseconds.
- If no page fault occurs, memory access time with paging is not much more than this.
- Let $p$ be the probability of a page fault on any memory access,
  where $0 \leq p \leq 1$
- Effective access time would be
  $$(1-p) \times ma + p \times \text{page fault time}$$
Servicing a Page Fault

- Service the page fault interrupt (1-100 μs)
- Read in the page (8ms seek, 4ms disk latency, 1ms transfer = 13ms)
- Restart the process (1-100 μs)
- Total time, say 14ms.

Access Time

- So the effective access time is 
  \[(1-p) \times 100\text{ns} + p \times 14\text{ms}\]
  \[(1-p) \times 100 + p \times 14\,000\,000 \text{ (in ns)}\]
  \[100 + 13\,999\,900 \times p\]
- Effective access time depends on page fault rate. E.g., if a maximum 10 percent degradation is required, then we require
  \[110 > 100 + 13\,999\,900 \times p\]
  \[10 > 13\,999\,900 \times p\]
  \[p < 0.000007\]

Page Replacement

- When there is no free frame, a page currently in memory must be paged out to allow a required page to come in.
- Page replacement algorithm determines which pages remain in memory for future use → impacts future page faults.

Page Replacement

- If free frame, use it.
- Select victim page (page replacement alg.)
- Write victim to disk.
  - Better: only write if page modified.
  - Dirty bit set when page modified – hardware support.
- Update victim page table.
- Read new page into victim frame.
- Update page table.
Demand Paging - Summary

- Demand paging completed the separation between logical and physical memory.
- A process with any number of pages can be fitted in to a lesser number of pages by page replacement.
- Implementation of demand paging requires a page replacement algorithm.

Page Replacement Algorithms

- Aim: low page fault rate.
  - Removing a page means that the next access to that page will be a page fault.
- An algorithm can be evaluated by running it on a particular string of memory references (called a reference string)
- Increased page frames usually means decreased page faults (but not linear)

**FIFO**

- The simplest page replacement algorithm is to find the page which has been in memory the longest and send it out to disk
  - We can do this without actually recording the time the page was loaded by placing the pages in a FIFO queue.
  - Replace page at head of queue.
  - Append new page to tail of queue.

Process has 3 frames
Reference string (length 17)

```
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0
```

12 page faults
Problem with FIFO

- A page may have been loaded a long time ago but still be frequently accessed.
  - E.g. the code for a loop over a very large array.

Belady's Anomaly

- We would expect that as the number of frames is increased, the number of page faults would decrease.
- Generally true, but not always.
- For some algorithms and particular reference strings the number of page faults increases if the number of frames is increased (at least for small increases).
- This is known as Belady's anomaly.

Optimal Algorithm

- Belady's discovery → search for optimal algorithm.
- Optimal algorithm: always replace the page which will not be used for the longest period of time.
- Requires perfect future knowledge.

Optimal algorithm

Process has 3 frames
Reference string (length 17)

8 page faults
Use of Optimal Algorithm

- Can’t use optimal algorithm for paging in an OS.
- Can use it to compare performance of practical algorithms.
- Most page replacement algorithms are an attempt to estimate which page which not be used for the longest period of time.

Approximating the Optimal Algorithm

- Optimal: Page not needed for longest time.
- FIFO: Oldest page in memory.
- LRU: Page not used for longest time.
  - Approximations to LRU:
    - Usage bits: Least recent usage pattern.
    - Second chance: Not referenced recently.
- LFU: Page used least frequently.
- MFU: Page used most frequently.

LRU

- by selecting the page which has not been used for the longest period of time we hope that this will be the page will not be used for the longest period in the future
- this is the least recently used (LRU) algorithm
- it gives good performance, but its implementation has problems
- the fundamental problem is how to order pages based on order of use

LRU algorithm

Process has 3 frames
Reference string (length 17)

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1

11 page faults
Implementing LRU

- LRU gives good performance.
- How to keep track of LRU page?
  - Time of use for each page in page table
    - Clock time or counter (easier)
  - Stack of page numbers
    - On page reference: remove from stack and place on top.
    - LRU page is always on the bottom of the stack.
  - Update on every reference: hardware support.

Notes on LRU

- LRU does not suffer Belady's anomaly.
- It is one of a class of page replacement algorithms, known as stack algorithms, that do not suffer Belady's anomaly.
- For these algorithms it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory for n+1 frames.

LRU Approximation Algorithms

- Few systems provide the hardware support needed for LRU.
- Many do provide a reference bit – set whenever page is accessed.
- Can determine which pages have been referenced and which have not.
- Use reference bits to approximate LRU.
  - Regularly read and reset the reference bits.
  - Record reference information.

Additional Reference Bits (Aging)

- Keep reference history.
- E.g. one byte per page.
- Regularly (e.g. every 100ms) shift bytes right and copy each reference bit into corresponding high bit of each byte.
- Records recent usage pattern. e.g 00011001
- Treat bytes as unsigned integers.
- Lowest value has least recent usage history.
Second Chance

- Enhance FIFO with a single reference bit.
- Basic algorithm is FIFO.
- Repeat until done:
  - Examine the FIFO queue head.
  - If reference bit is clear, replace that page.
  - If reference bit is set:
    - Reset reference bit.
    - Move page to the tail of FIFO queue.

Enhanced Second Chance (Not Recently Used)

- Use reference and modified bits.
- Periodically reset reference bits.
- Select a page from best class in order of preference using (Ref,Mod) bits:
  - (0,0) not used nor modified
  - (0,1) not used recently, but modified, so would have to be written out
  - (1,0) used but not modified – no write required
  - (1,1) used and modified

LFU & MFU

- Count uses of page instead of tracking time.
- LFU: Replace least used page (unlikely to be needed).
- MFU: Replace most used page (likely to be finished with it).
- These algorithms are expensive to implement and do not give a good approximation of optimal performance.
Page Buffering

- Reduces I/O delay for page replacement.
- Keep at least one frame free.
- Select a victim as before.
- Read new page into free frame so it does not have to wait for victim to be unloaded.
- Unload the victim to maintain free frame collection.

Pool of free frames

- As an extension to the pool of free frames (page buffering), a record can be kept of which page was in each of the free frames;
- if those pages need to be read back in before the frames are re-used no actual I/O need be carried out.

flushing

- From time to time, when backing storage device is idle, write out modified frames.
- Reduces the chance of needing to write out victim when it is replaced.

Frame Allocation

- How many frames should be allocated to each process?
- Limitations:
  - Minimum number of frames which can be allocated is defined by the architecture
  - Maximum is defined by the size of physical memory
  - Total number of frames allocated to all processes cannot exceed the total number of frames in the actual memory
Minimum Allocation

- Undesirable: very few pages → many page faults.
- Minimal useable number of pages:
  - All pages used by an instruction must be in memory.
  - If not in memory, page fault and retry.
  - So: minimum is maximum pages that a single instruction can require.
  - E.g. load/store architecture: 2

Indirection

- For some architectures, a single instruction can access up to nine frames
- For architectures which allow multiple levels of indirection (using one bit in the address to indicate whether it is an indirection or not) the worst case is the entire virtual memory
- Such architectures must limit the number of levels of indirection

Allocation Algorithms

- Equal allocation: evenly divide frames amongst the processes, with any left over forming the free pool
- Processes vary significantly in size, so this is probably not the best solution
- Proportional allocation, where processes of larger size are given more frames, proportional to their size

Notes on Allocation

- Number of frames allocated to any process will depend upon the degree of multiprogramming
- Variant on proportional allocation: use not the process' size, but their priority, or some combination of the two
Global versus Local Allocation

- **Global**: when selecting a page to replace, choose from any frame in memory.
- **Local**: replace only pages belonging to the process.
  - Number of frames allocated to a process does not change

Global Replacement

- With global replacement, the page fault rate of a process can be adversely affected by other processes claiming its frames.
- However, it may claim them back and local allocation prevents processes servicing page faults at the expense of the least used pages in memory.
- For this reason global allocation is the most commonly used method.

Page Faults

- If the number of pages allocated to a process falls below the minimum required by the architecture then all its remaining page(s) must be paged out.
- If a process has more frames than the minimum, but less than it currently wants to use it will generate frequent page faults.
- If local allocation is being used, the fault will replace one of the process' own pages.
  - But that page was also in current use, so another fault will soon result.

Thrashing

- A process is *thrashing* if it is spending more time page faulting than executing.
- Thrashing can result in severe performance problems.
- When CPU utilisation is low, most systems will increase the degree of multiprogramming.
- As more and more processes are brought into the system the demand for page frames increases.
Results of Thrashing

- This will push more processes below their working (as opposed to architecture imposed) minimum and more page faults will result
- This will push the CPU utilisation further down, more processes will be introduced and the system will enter a downward spiral

Prevention of Thrashing

- To prevent thrashing the system needs to know how many frames each process needs
- There are several techniques to determine this quantity

Thrashing

![Thrashing Diagram](image)

Locality Model

- The *locality model* is an attempt to determine how many frames a process is using
- The locality model states that, as a process executes, it moves from locality to locality
- A locality is a set of page that are used together, often consisting of a procedure and the data it accesses
- A program is composed of a number of localities, parts of which may overlap
- If enough pages frames are allocated to hold the current locality then the process will not thrash while it is in that locality
Working Set Model

- the working set model is based on the assumption of locality
- it uses a parameter, \( d \), to define the working set window
- the set of pages accessed in the \( d \) most recent page references is the working set
- a page will drop from the working set \( d \) time units after its last reference
- the working set is an approximation of the program's locality

Working Set Model

- if \( d \) is too small it will not encompass an entire locality
- if \( d \) is too large it will reduce the possible degree of multiprogramming
- if we can compute the size of the working set for each process then we can calculate the sum of these sizes, and this is the total demand for frames, \( D \)
- if \( D \) is greater than the number of frames than thrashing will occur
- processes should be suspended until \( D \) falls below the number of frames

Process Initialisation

- a process may only be initiated if enough frames are free so that the new process will not push \( D \) over the limit
- otherwise it must wait

Problems with Working Set

- the problem with the working set model is keeping track of the working set
- it must be updated on each reference
- the working set can be approximated by use of a fixed time interval interrupt and a reference bit for each page
Page Fault Frequency

- A more direct way to control thrashing is to examine the frequency of page faults.
- After all, the essential problem with thrashing is too high a page fault rate.
- Set upper and lower bounds on the acceptable page fault rate – determine when the process should be given more or less pages frames.
- This may then affect the degree of multiprogramming.

Prepaging

- With pure demand paging, or when a process is swapped out, its first (or next) bout of execution will result in a number of page faults until the required locality is brought into memory.
- Preparing is an attempt to reduce this by bringing in as many pages at once as possible.
- When a process is swapped out, a record of its working set is kept.
- When it is to be brought back in, its entire working set can be placed in memory before the process is restarted.

Page Size

- When designing an operating system for an existing machine, the page size is set.
  - But some architectures support multiple page sizes.
- When designing a specification for a new machine, page size can be chosen.
- One consideration is the relationship between the page size and the sector size on the paging device.

Page Size

- Page size is always a power of two, so that all its addresses can be accessed by using all values of a given number of bits.
- Decreasing the page size increases the size of the page table but gives better memory utilisation.
- A larger page size reduces I/O time.
- It is quicker to read in one page of a given size than two pages half the size, as latency and seek time are much greater than actual transfer time.
**Page Size**

- smaller page size may give better locality by bringing in only the information required, not that and more
- however, smaller page size may result in more page faults
- page sizes are increasing, as the cost of servicing a page fault is regarded as less attractive than the resulting internal fragmentation

**Program Structure**

- the number of page faults can be reduced by careful structuring of the code and data of a program
- for example, assume a page size of \( n \) words and an \( n \times n \) array assume each row is stored on one page
  - if the array is accessed row by row then one page is dealt with before the next and so on
  - if the array is accessed column by column then one word on each page is accessed \( \rightarrow \) page faults
- while paging is meant to be transparent to the users, an understanding of it can be used to write programs in such a way that the page fault rate is lowered

**I/O Interlock**

- when demand paging is used we may require the continued presence of some pages in memory
- when a process requests some I/O, a page (or pages) will be involved
- the addresses will be passed to a controller, which will be responsible for the I/O
- the process will then be blocked, waiting for the I/O to complete
- the pages should not be paged out in the meantime, or the controller will overwrite the information of another process

**First Solution**

- I/O can never be executed to user memory
- I/O takes place between system memory and the devices
- data is then copied between system and user memory
- the extra copying may become an unacceptable overhead
**Second Solution**

- another solution is to allow pages to be *locked* into memory
- a locked page cannot be replaced - so I/O can easily involve the page
- the lock bit can also be used to hold pages in memory for processes which are in the ready queue
- this can be dangerous, as the frame may be needed for other processes
- also, if there are any failings in the system, lock bits may be turned on and never turned off, rendering the frame useless

**Real-Time Processing**

- due to the unexpected delays which can be caused by virtual memory hard real time systems almost never use it
- some soft-real time systems allow processes to give "hints" to the system which specify those pages that are important to the process
- this should only be available to privileged processes, or the whole system could be locked up

**Virtual Memory - Summary**

- demand paging can be used to allow a program’s storage to be larger than main memory
- memory must be divided amongst the processes
- each process will have only some of its pages in memory
- a number of algorithms can be used to select a page to get rid of when another page must be brought in
- the optimal algorithm cannot be implemented

**Virtual Memory - Summary (cont.)**

- thrashing occurs when most processes have insufficient pages
- demand segmentation is similar in principle to demand paging, but is less efficient while requiring less hardware support